ANALYSIS OF MINIMIZATION ALGORITHMS FOR MULTIPLE-VALUED PROGRAMMABLE LOGIC ARRAYS*

Parthasarathy Tirumalai
52L/57, Hewlett-Packard Company
5301 Stevens Creek Boulevard
Santa Clara, CA 95051

Jon T. Butler
Dept. of Electr. and Comp. Eng.
Naval Postgraduate School
Monterey, CA 93943-5100

ABSTRACT

We compare the performance of three heuristic algorithms [3,6,13] for the minimization of sum-of-products expressions realized by the newly developed multiple-valued programmable logic arrays [9]. Heuristic methods are important because exact minimization is extremely time-consuming. We compare the heuristics to the exact solution, showing that heuristic methods are reasonably close to minimal. We use as a basis of comparison the average number of product terms over a set of randomly generated functions. All three heuristics produce nearly the same average number of product terms. Although the averages are close, there is surprisingly little overlap among the set of functions where the best realization is achieved. Thus, there is a benefit to applying different heuristics and then choosing the best realization.

I. INTRODUCTION

The minimization of sum-of-products expressions in binary logic has received considerable attention for over 30 years. The complexity of the problem has been known for almost as long. Although minimal sum-of-products extraction appears to be a special case of the general minimal set covering problem, it is not. This was proven by Gimpel [7] in 1965. That is, any instance of the set covering problem is an instance of the minimal sum-of-products extraction. In 1972, Karp[8] showed that the set covering problem is NP complete; thus, so also is minimal sum-of-products extraction. The best known algorithm then requires exponential time. This is a real barrier; it precludes the exact minimization of functions with even a moderately low number of inputs, e.g. 20. As a result, considerable effort has been devoted to heuristic minimization methods. For example, among the Berkeley VLSI tools is ESPRESSO-IIC [4], a C program that minimizes binary functions by a set of operations on the prime implicants.

Recently, there has been considerable interest in multiple-valued PLA's [1,2,3,5,6,9,10,11,13,14,15]. Several have been proposed [11,14,15] and at least one implemented [9]. We know of three heuristic MVL sum-of-products minimization algorithms. Pomper and Armstrong [13] introduced in 1981 a heuristic method that found a near-minimal sum-of-products expression as a direct cover of the function. The algorithm proceeds in two steps; 1. select a minterm and 2. find an implicant that covers the minterm. The first step is accomplished by a *random* choice of minterm, while the second step is accomplished by choosing the *largest* implicant. In 1986, Besslich [3] introduced another direct cover method that seeks to cover the "most isolated" minterms first. And in 1987, Dueck and Miller [6] introduced a method which also seeks the most isolated minterm first, but chooses a product term that tends to introduce the fewest discontinuities when subtracted from the function.

There has been little study of the relative merits of heuristic algorithms. To the credit of Brayton, Hachtel, McMullen and Sangiovanni-Vincentelli [4], the realizations produced by ESPRESSO-IIC were compared with the realizations of MINI, PRESTO, and POP [4] over a set

*Research supported in part by NSF Grant MIP-8706553, in part by NATO Grant 423/84, and in part by an NPS Foundation Grant.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **MAY 1988**	2. REPORT TYPE	3. DATES COVERED
4. TITLE AND SUBTITLE **Analysis of Minimization Algorithms for Multiple-Valued Programmable Logic Arrays**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Postgraduate School,Department of Electrical and Computer Engineering,Monterey,CA,93943**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

We compare the performance of three heuristic algorithms [3,6,13] for the minimization of sum-of-products expressions realized by the newly developed multiplevalued programmable logic arrays [9]. Heuristic methods are important because exact minimization is extremely time consuming. We compare the heuristics to the exact solution, showing that heuristic methods are reasonably close to minimal. We use as a basis of comparison the average number of product terms over a set of randomly generated functions. All three heuristics produce nearly the same average number of product terms. Although the averages are close, there is surprisingly little overlap among the set of functions where the best realization is achieved. Thus there is a benefit to applying different heuristics and then choosing the best realization.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**11**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

of 56 specifically chosen binary functions. However, the justification of the newly introduced algorithms examined by us has rested on an intuitive notion, supported by examples. In this paper, we analyze six synthesis methods.

1. Random minterm random implicant
2. Pomper and Armstrong [13]
3. Besslich [3]
4. Dueck and Miller [6]
5. Gold
6. Absolute minimization

over a set of 7000 randomly chosen 2-variable 4-valued functions. Our choice of 2-variable functions is influenced by a need to compare heuristics (1-5) with absolute minimal realizations (6). Even with only 2 inputs, absolute minimization requires high computation times, on the order of days. Our choice of 4-values is influenced by the current interest in this radix. The three heuristic algorithms in [3,6,13] apply to sum-of-products expressions where sum is MAX. In all three algorithms, the case where sum is truncated SUM is also considered. Because the PLA's produced in CCD [9] use the SUM operation, our analysis uses this operation. Further, we have adapted the heuristics to the case of product terms consisting of interval literals, again since the CCD PLA's realize such functions. A motivation for the study reported here is the development of a minimization method for our CCD PLA CAD tool [10].

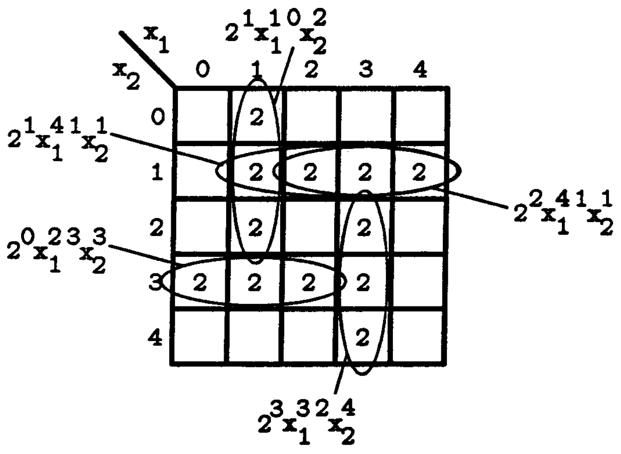
This paper is organized as follows. The next section introduces notation and fundamental concepts. Section III describes the six synthesis methods, and Section IV describes the results of a comparative analysis of their performance. In the final section, we summarize the results.

II. BACKGROUND AND NOTATION

Let $X = \{x_1, x_2, \cdots, x_n\}$ be a set of n variables, where x_i takes on values from $R = \{0, 1, ..., r-1\}$. A function $f(X)$ is a mapping $f: R^n \rightarrow R \bigcup \{r\}$, where r is the *don't care* value. Specifically, $f(X)$ is said to be an *n-variable r-valued function.* Fig. 1 below shows a map representation of a 2-variable 5-valued function. Blank entries correspond to 0. An assignment x of values to variables in X is called a *minterm* if $f(x) \neq 0$. In Fig. 1, there are 12 minterms all of which yield a 2.

Functions realized by the PLA's described in Kerkhoff and Butler [9] are composed of three functions,

1. MIN: $f(x_1,x_2) = x_1 x_2$ $(= MIN(x_1,x_2))$,
2. SUM: $f(x_1,x_2) = x_1 + x_2$ $(= x_1 + x_2$ if $x_1 + x_2 \leq 3$ and $= 3$ otherwise, where $+$ is viewed as ordinary addition and x_i is viewed as an integer), and
3. literal: $f(x_1) = {}^a x_1^b$ $(= r-1$ if $a \leq x_1 \leq b$ and $= 0$, otherwise).



x_1
x_2
$2\,{}^1x_1^1\,{}^0x_2^2$
$2\,{}^1x_1^4\,{}^1x_2^1$
$2\,{}^2x_1^4\,{}^1x_2^1$
$2\,{}^0x_1^2\,{}^3x_2^3$
$2\,{}^3x_1^3\,{}^2x_2^4$
0
1
2
3
4

Figure 1. Example of a 2-variable 5-valued function.

In binary, the SUM, MIN, and literal correspond to AND, OR, and x^*, where $x^* \in \{x, \bar{x}\}$. In the realization of functions by a multiple-valued PLA, constants and literals occur as operands of the MIN functions. A *product term* is the MIN of one nonzero constant and one or more literals. For example, $f(x_1,x_2) = 2\,{}^2x_1^4\,{}^1x_2^1$ is a product term that is 2 when x_1 is 2, 3, or 4 and x_2 is 1. An *implicant* of function $f(X)$ is a product term $I(X)$ such that $f(X) \geq I(X)$. A *prime implicant* of $f(X)$ is an implicant of $f(X)$ such that there is *no* other implicant $I'(X)$ of $f(X)$ where $I'(X) \geq I(X)$. For example, $2\,{}^2x_1^4\,{}^1x_2^1$ is an implicant of the function in Fig. 1, but it is not a prime implicant. However, $2\,{}^1x_1^4\,{}^1x_2^1$ is a prime implicant. Any function can be expressed as the SUM of implicants [15]. For example, the function $f(x_1,x_2)$ in Fig. 1 can be expressed as the SUM of 4 implicants,

$$2\,{}^2x_1^4\,{}^1x_2^1 + 2\,{}^3x_1^3\,{}^2x_2^4 + 2\,{}^0x_1^2\,{}^3x_2^3 + 2\,{}^1x_1^1\,{}^0x_2^2. \quad (1)$$

We use the term *sum-of-products* to describe functions realized by such PLA's, where sum refers to SUM and product refers to MIN. A sum-of-products expression for function $f(X)$ is *minimal* if there is no other expression for $f(X)$ with fewer product terms. For example, (1) is a minimal sum-of-products expression. Given $f(X)$, implicant $I(X)$ *covers* a minterm at $\boldsymbol{x}$ if $f(\boldsymbol{x}) = I(\boldsymbol{x})$. Therefore, $g(X) = f(X) - I(X)$ has the property $g(\boldsymbol{x}) = 0$, and we say that subtracting $I(X)$ *drives* the minterm at $\boldsymbol{x}$ to 0.

It is interesting to note that the *none* of the implicants in (1) are prime. In fact, any sum-of-products expression for $f(x_1,x_2)$ containing at least one prime implicant is *not* minimal. Such expressions require at least 5 product terms. Therefore, in contrast to the case of conventional binary sum-of-products, it is necessary to consider all implicants. A similar observation was made with respect to sum-of-products expressions, where sum is MAX and products are the MIN of unary functions of varying costs [12].

III. MINIMIZATION ALGORITHMS FOR SUM PLA'S

We consider six synthesis methods. The first five are heuristic and are based on the direct cover approach. In this approach, a minterm is first chosen. Next, an implicant is chosen that covers this minterm. The implicant is then subtracted from the function and the process is repeated until there are no more minterms.

1. Random Minterm Random Implicant

In this heuristic, both the minterm and implicant are chosen randomly, with all choices equally likely. That is, among all minterms, one is chosen randomly, and similarly in the case of implicants. As with the other algorithms, the chosen implicant is subtracted and the process repeated on the resulting function. Since no particular characteristics of the function are used to determine the choice of minterm or implicant, it provides a basis of comparison for the next four heuristics, which use characteristics of the functions to limit, to varying degrees, the choices of minterms and implicants.

2. Pomper and Armstrong

As in the Random Minterm Random Implicant heuristic, this heuristic chooses the minterm randomly. However, the implicant is chosen as the one, which when subtracted, drives the largest number of minterms to 0 or don't care. If there is more than one such implicant, the largest is chosen. If there are more than one largest, the one generated first is chosen. The process is repeated until the function is completely covered. A formal description of this algorithm is given in Appendix I.

Because the Random Minterm Random Implicant heuristic can choose from all possible covers of any function, there is a nonzero probability that it will find a minimal realization. However, if the minimal realizations are a small percentage of the total number of realizations, non-minimal realizations will most likely result. Our analysis shows this clearly. There is then the question of whether the Pomper and Armstrong heuristic has a nonzero probability of finding a minimal realization for all functions. The example in Fig. 1 is a function where this probability is 0. Here, a prime implicant is never in a minimal sum-of-products realization, and so, for this case, the Pomper and Armstrong heuristic does *not* produce a minimal realization.

3. Besslich

Besslich [3] presents a direct cover approach for the heuristic minimization of multiple-valued logic functions using minterm weighting and implicant detecting transformations. In this heuristic, each minterm is assigned a weight that is a measure of the degree to which other minterms cluster around it. Minterms in the center of clusters have the highest weight and isolated minterms the lowest. The minterm with the smallest weight is chosen. Next all implicants that cover the chosen minterm are generated. For each, an efficiency factor equal to the cost of the implicant divided by the number of minterms it covers is calculated and the minterm with the largest factor is chosen. In a PLA, all implicants are realized with the same cost (one column). Therefore, for this case, only the number of minterms driven to 0 or don't care determines which implicant is chosen. The basic idea of the Besslich heuristic is to cover the most isolated minterms first and use implicants that have a low cost per minterm covered. Besslich [3] does not mention of how ties are broken. We choose to break ties among implicants having the same efficiency factor by using only the largest implicants, and among these by choosing the first one generated. Thus, the choice of implicants is made in the same way as in the Pomper and Armstrong heuristic. A formal description of the algorithm is given in Appendix II.

4. Dueck and Miller

Dueck and Miller [6] present a heuristic similar to that of Besslich's with the intent of improving realizations using the SUM operation. In this approach, an isolation factor (IF) is calculated for each minterm with the smallest value (that is, all 1 minterms are considered first; if there are none, then 2 minterms are considered, etc.). The isolation factor of a minterm at x is inversely proportional to 1 plus the number of *adjacent* minterms plus the number of logic variables where there are a nonzero number of such minterms. Similar to the weight transform presented by Besslich, the isolation factor provides a measure of the degree to which a specific minterm can combine with other minterms in the function (the correlation is negative, however). The minterm with the highest isolation factor is chosen. All implicants that cover this minterm are then generated. For each of these, a parameter called the relative break count (RBC) is calculated. This provides a measure of the degree to which the function is simplified if the implicant under consideration is chosen. The idea is to judiciously choose implicants that make the remaining function easy to realize. Our use of interval literals requires an adjustment to the Dueck and Miller algorithm. A formal description of our algorithm, modified to the specifications of our problem, is given in Appendix III.

5. Gold

Gold is a heuristic in which the heuristic algorithms of Pomper and Armstrong, Besslich, and Dueck and Miller are applied and the best realization is chosen. It was inspired by the observation that these algorithms displayed a diversity in realizations. That is, no single algorithm is consistently better than the others over all functions; there are classes of functions where one algorithm does better than the others.

6. Absolute Minimization

An algorithm which produces the exact minimal sum-of-products was devised to compare with the results produced by the previous four algorithms. This does essentially an exhaustive search over all possible solutions, starting with the fewest number of product terms. The algorithm requires considerable computer time. A complete description of the algorithm is given in Section VI.

7. Summary

Table I below summarizes the six algorithms.

Algorithm	Choice of Minterm	Choice of Implicant
1. Random Min. Random Impli.	Random	Random
2. Pomper and Armstrong [13]	Random	Drives most min. to 0
3. Besslich [3]	Smallest weight	Drives most min. to 0
4. Dueck and Miller [6]	Largest IF	Smallest RBC
5. Gold	Best of 2, 3, and 4	
6. Absolute Minimization	Exhaustive search	

Table I. Summary of Minimization Algorithms

IV. COMPARISON OF ALGORITHM PERFORMANCE

For the purpose of comparison, we separate all four-valued, two-variable functions into 17 disjoint classes according to the number of non-zero values in the function. For 14 of the 17 classes, 500 random functions are generated. For each function, the six algorithms are applied and the number of product terms derived. For functions with less than three non-zero values, the average number of product terms can be calculated as follows.

0 non-zero values: There is only one (trivial) function, constant zero, in this class. This function requires no implicants in its cover, and all algorithms produce identical results.

1 non-zero value: There are 48 functions in this class, and each requires uniquely one implicant. Thus, the average number of implicants required by all algorithms is 1.

2 non-zero values: There are $3^2 \cdot \binom{16}{2} = 1080$ such functions. All require two implicants unless there are two adjacent minterms with the same value, in which case a single implicant is sufficient. Of the 1080 functions, 72 can be covered by one implicant. Only the Random Minterm Random Implicant algorithm will fail to find the minimal cover for all functions in this class. Specifically, functions with adjacent identical minterms will be incorrectly minimized half the time, since a cover of a single minterm or two adjacent minterms are equally likely. For a uniformly distributed set of functions, the average and standard deviation on the number of implicants produced by the Random Minterm Random Implicant algorithm are 1.97 and 0.18, respectively. On the other hand, these values for all other algorithms are 1.93 and 0.25, respectively. Our random function generation program was tested on this class and yielded values very close to these, to within 0.18% for the average and to within 3.9% for the standard deviation.

For classes with a larger number of non-zero values, 500 functions were randomly generated. The functions generated had no don't care values. However, they could develop don't care values during the covering process. (Nota bene: A value of 3 in the original function becomes a don't care when fully covered.) Fig. 2 shows the results.

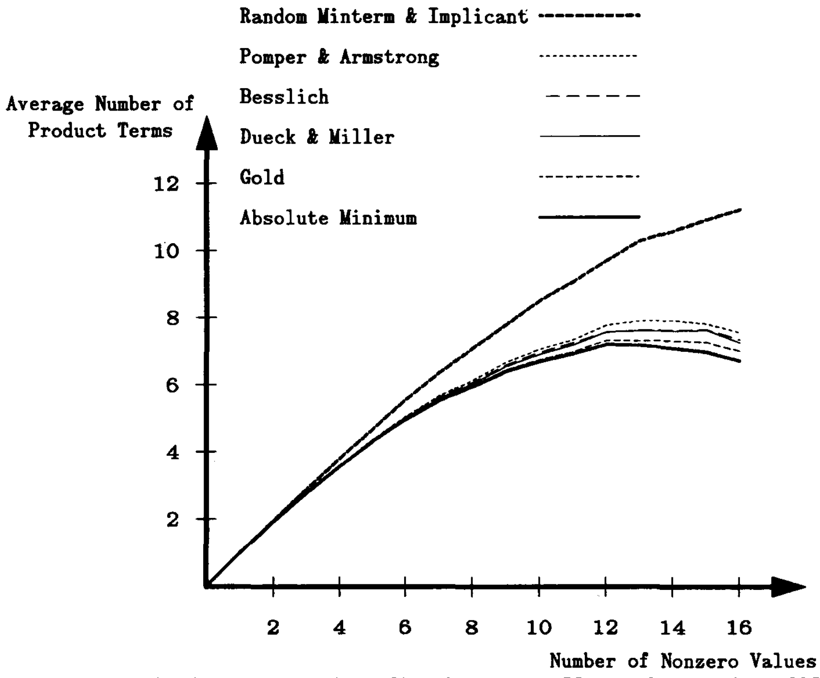


Figure 2. The Average Number of Product Terms Verses the Number of Nonzero Values for the Six Algorithms.

It can be seen that the Random Minterm Random Implicant algorithm does quite poorly. Choosing the largest (prime) implicant, as in the Pomper and Armstrong algorithm, provides a significant improvement. There is very little difference between the performance of the Besslich and the Dueck and Miller algorithms; their curves in Fig. 2 are almost identical. Both algorithms choose the most isolated minterm first, and this gives some improvement over that of the Pomper and Armstrong algorithm. Gold, taking advantage of the small overlap between the three heuristic algorithms, provides some further improvement. Still further improvement is achieved by absolute minimization, shown as the lowest curve in the figure.

Another measure of comparison of the algorithms is the number of functions for which the absolute minimal solution is found. Fig. 3 shows how the six algorithms compare on this basis. Here there is a much larger distinction between the algorithms. As the number of non-zero values increases, the number of functions for which an optimum cover is found by the Random Minterm Random Implicant algorithm drops off sharply, to nearly zero. The Pomper and Armstrong algorithm offers considerable improvement, but even in this case only about 39% of the functions are completely minimized when the number of non-zero values is 14-16. The Besslich and Dueck and Miller algorithms are again very close to each other. Approximately, 52% of the functions are minimized for these two algorithms when the number of non-zero values is 14-16. Gold shows a reasonable improvement, minimizing about 75% of the functions with 14-16 non-zero values.

Tables II and III show the numerical values obtained from the programs. Another run was made using different seeds for the random number generator streams. The results from both runs were very similar, indicating that a sufficiently large sample set size was used. An examination of the data suggests the truth of the following.

Conjecture: No more than 10 implicants are needed in a minimal sum-of-products expression of a four-valued, two-variable function, where sum is the SUM operation and the products consist of the MIN of a constant and interval literals on the variables.

The absolute minimization program and the

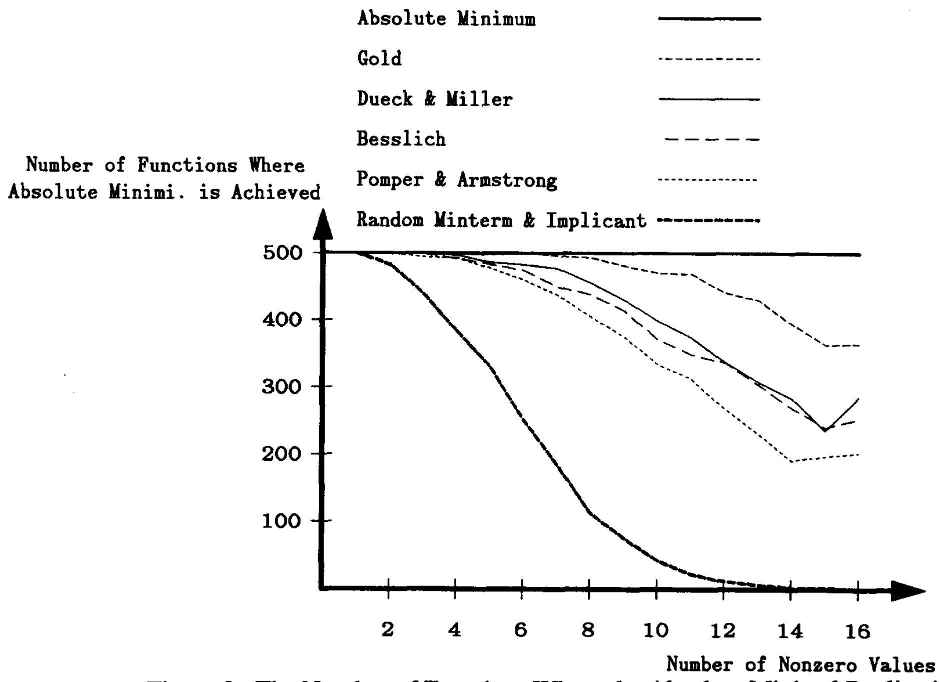


Figure 3. The Number of Functions Where the Absolute Minimal Realization is Achieved Verses the Number of Nonzero Values.

Number of Nonzero Values	Random Minterm Random Implicant	Pomper Armstrong	Besslich	Dueck Miller	Gold	Absolute Minimiza.
0	0.00	0.00	0.00	0.00	0.00	0.00
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.97	1.93	1.93	1.93	1.93	1.93
3	2.90	2.79	2.78	2.78	2.78	2.78
4	3.80	3.58	3.58	3.57	3.57	3.57
5	4.66	4.35	4.34	4.33	4.31	4.31
6	5.53	5.04	5.02	5.00	4.96	4.96
7	6.31	5.66	5.63	5.57	5.53	5.52
8	7.04	6.13	6.07	6.03	5.95	5.94
9	7.77	6.67	6.59	6.55	6.44	6.40
10	8.50	7.06	6.97	6.91	6.75	6.69
11	8.99	7.34	7.25	7.19	6.98	6.92
12	9.78	7.77	7.56	7.57	7.32	7.20
13	10.31	7.90	7.63	7.60	7.31	7.17
14	10.54	7.91	7.62	7.60	7.30	7.08
15	11.00	7.81	7.64	7.63	7.26	6.97
16	11.21	7.55	7.33	7.24	6.99	6.71

Table II. Average Number of Product Terms

Number of Nonzero Values	Random Minterm Random Implicant	Pomper Armstrong	Besslich	Dueck Miller	Gold	Absolute Minimiza.
0	500	500	500	500	500	500
1	500	500	500	500	500	500
2	483	500	500	500	500	500
3	439	494	499	499	500	500
4	384	493	492	497	499	500
5	331	477	483	486	497	500
6	252	460	474	483	500	500
7	187	438	450	477	496	500
8	113	405	438	455	493	500
9	76	375	414	430	481	500
10	43	334	371	399	470	500
11	23	313	349	374	469	500
12	12	268	337	338	441	500
13	7	231	304	308	430	500
14	3	191	269	284	394	500
15	3	197	240	235	363	500
16	1	202	253	285	365	500

Table III. Number of Functions Where Absolute Minimi. is Achieved.

random function generation routines were written in C. The heuristic algorithms were written in Pascal. The program was executed on a Hewlett-Packard Series 9000 Model 350 workstation running HP-UX (HP's extension of the UNIX operating system). The program took more than 3 days to minimize 7000 randomly generated functions with 3 to 16 non-zero values.

V. ABSOLUTE MINIMIZATION.

The program to compute the absolute minimal realization of a function is a search over combinations of implicants. As observed in Section II, it is necessary to consider all implicants and not just the prime implicants. In the worst case for two-variable four-valued functions ($f(x_1,x_2) = 3$), this can be large, 300 implicants. Consider a function which has 100 implicants and requires 8 product terms in a minimal cover. Using a brute force enumeration technique, the proof that this is the minimum, requires that all combinations of seven or fewer implicants be examined. There are more than 17 billion such combinations, and assuming that each combination can be generated and examined in 200 microseconds (a value that is better than in our program), it would take more than 944 hours, or 39 days, to examine all the combinations for one function! Clearly this is not acceptable. However, using the lemma given below, the search tree can be significantly reduced.

Definition: $U(x)$ is a *constrained implicant set* of minterm x for function $f(X)$, if $U(x) = \bigcup_{I(x)>0} I(X)$, where $I(X)$ is an implicant of $f(X)$.

Lemma 1: If $U(x)$ is a constrained implicant set, then every minimal sum-of-products expression of $f(X)$ contains at least one implicant from $U(x)$.

The proof follows from the fact that at least one implicant is needed from each of the k disjoint sets.

Definition: $U(x)$ is a *minimal* constrained implicant set of a function if and only if $0 < |U(x)| \leq |U(y)|$ for all $y \in X$, where $U(x)$ is a constrained implicant set.

A minimal constrained implicant set is useful in an efficient search for the absolute minimal sum-of-products expression for a function. The search space can be represented by a tree where each node corresponds to a function, and each edge to an implicant. The root node represents the input function, and all other nodes represent functions obtained by subtracting from the input function, the implicants in the path (from this node) to the root. If a function has ϕ implicants, the root has ϕ descendents. Further, each of these has at most (and probably less than) $\phi - 1$ descendents. However, rather than consider all implicants at the root node, it is sufficient to consider only implicants from a minimal constrained implicant set, considerably reducing the search space. From Lemma 1, at least one of this set must be in a minimal sum-of-products expression. For example, the average number of implicants for a four-valued, two-variable function with one zero, (obtained from a randomly generated sample set of 500 functions) is 111.2, while the average size of the minimal constrained implicant set is only 7.9. This is a considerable reduction. In fact, without it the computation of the absolute minimum algorithms would be impossible.

The algorithm recursively finds the absolute minimum cover of the function at each node. Let $G(f)$ denote the minimum number of implicants needed in the minimal sum-of-products expression of function $f(X)$. Let $U(x) = \{I_1(X), I_2(X), \ldots, I_k(X)\}$ be any constrained implicant set of $f(X)$. Let $g_i(X) = f(X) - I_i(X)$ be the function obtained by subtracting $I_i(X)$ from $f(X)$. From the definition of a constrained implicant set, it follows that, $G(f) = 1 + \min\{G(g_1), G(g_2), \ldots, G(g_k)\}$.

ABSOLUTE MINIMIZATION ALGORITHM

```
f ← the input function;
cur_best_soln_size ← M {see below};
cur_soln_size ← 0;
if f has no coverable minterms, then
                    output 0 and stop;
recursively_minimize(f);
output cur_best_soln_size;
stop;

procedure recursively_minimize(f);
U ← some constrained implicant set of f;
while ((there exists another implicant in U)
      AND ((cur_soln_size + 1) <
      (cur_best_ soln_size))) do begin
   I ← the next implicant in U;
```

```
        add I to the current solution;
        cur_soln_size ← cur_soln_size + 1;
        g ← f - I;
         if g has no coverable minterms then
              begin
           make current solution set the current
             best solution set;
           cur_best_soln_size ← cur_soln_size;
        end;
        else  if  ((cur_soln_size  +  1)  <
                   (cur_best_soln_size))  then
                   begin
           recursively_minimize(g);
        end;
        delete I from the current solution;
        cur_soln_size ← cur_soln_size - 1;
end;
```

M represents the starting value for the solution size and is chosen as the best of the Pomper and Armstrong, Besslich, and Dueck and Miller solutions.

The speed of the absolute minimization algorithm depends on the choice of the constrained implicant set. When $U(x)$ is small, there are fewer choices for an implicant to cover x, and thus less time is required. This suggests that all possible sets be scanned and the smallest chosen. Although this reduces the *total* number of implicants scanned, the extra time required at each node may increase the *overall* program execution time. A heuristic was used to decide whether to spend this extra time or not. In our implementation, $U(x)$ was first generated by taking a coverable minterm in the function. An improvement was attempted only if the number of implicants in this set was greater than some threshold T. Values of 6 and 9 were tried with the former yielding a faster program.

VI. CONCLUDING REMARKS

There are three surprising results from this study. First, none of the three heuristic sum-of-products minimization algorithms [3,6,13] is clearly superior to the others. Second, Gold, an algorithm that picks the best of the three, does significantly better than any particular one. This indicates that the nearly identical average case performance is due to a diversity of performance on individual functions. Third, the computation times for absolute minimization far exceeded our expectation. Thus what began as an investigation of heuristic methods ended as an investigation of methods to reduce the search required by absolute minimization.

REFERENCES

[1] E. A. Bender and J. T. Butler, "On the size of PLA's required to realize binary and multiple-valued functions," forthcoming *IEEE Trans. on Comput.*, Jan. 1989.

[2] E. A. Bender, J. T. Butler, and H. G. Kerkhoff, "Comparing the SUM with the MAX for use in four-valued PLA's," *Proceedings of the 15th International Symposium on Multiple-Valued Logic*, May 1985, pp. 30-35.

[3] P. W. Besslich, "Heuristic minimization of MVL functions: A direct cover approach," *IEEE Trans. on Comput.*, February 1986, pp. 134-144.

[4] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic, Boston 1984.

[5] J. T. Butler and H. G. Kerkhoff, "Analysis of input and output configurations for use in four-valued programmable logic arrays," *Proceedings of the IEE-E: Computers and Digital Techniques*, July 1987, pp. 168-176.

[6] G. W. Dueck and D. M. Miller, "A direct cover MVL minimization using the truncated sum," *Proc. of the 17th Inter. Symp. on Multiple-Valued Logic*, May 1987, pp. 221-227.

[7] J. F. Gimpel, "A method of producing a Boolean function having an arbitrarily prescribed prime implicant table," *IEEE Trans. on Electron. Comput.*, June 1965, pp. 485-488.

[8] R. M. Karp, "Reducibility among combinatorial problems," in R. E. Miller and J. W. Thatcher, *Complexity of Computer Computations*, Plenum Press, New York 1972, pp. 85-103.

[9] H. G. Kerkhoff and J. T. Butler, "Design of a high-radix programmable logic array using profiled peristaltic charge-coupled devices," *Proceedings of the 16th International Symposium on Multiple-Valued Logic*, May 1986, pp. 100-103.

[10] H. G. Kerkhoff and J. T. Butler, "A module compiler for the design of high-radix CCD PLA's," preprint.

[11] H.-L. Kuo and K.-Y. Fang, "The multiple-valued programmable logic array and its application in modular design," *Proc. of the Inter. Symp. on Multiple-Valued*

Logic, May 1985, pp. 10-18.

[12] D. M. Miller and J. C. Muzio, "On the minimization of many-valued functions," *Proc. of the Inter. Symp. on Multiple-Valued Logic*, May 1979, pp. 294-299.

[13] G. Pomper and J. A. Armstrong, "Representation of multivalued functions using the direct cover method," *IEEE Trans. on Comput.* Sept. 1981, pp. 674-679.

[14] T. Sasao, "On the optimal design of multiple-valued PLA's," *Proc. of the Inter. Symp. on Multiple-Valued Logic* May 1986 pp. 214-223.

[15] P. Tirumalai and J. T. Butler, "On the realization of multiple-valued logic functions using CCD PLA's," *Proceedings of the 1984 International Symposium on Multiple-Valued Logic*, May 1984, pp. 33-42.

APPENDIX I

POMPER AND ARMSTRONG [13] ALGORITHM USING INTERVAL LITERALS

Orig_f is the given r-valued n-variable function. f is an intermediate function as it is covered in successive stages of the algorithm. Each minterm in f can have a value in the range [0...r] where r is the radix. A value of -

0 was zero in orig_f or was in the range [1...(r-2)] in orig_f but has now been fully covered,

1,2,...,(r-1) still needs to be covered,

r (don't care) was r in orig_f, or was (r-1) in orig_f but has now been fully covered.

Algorithm:

```
1. f ← orig_f
2. If f has no minterms in the range [1...(r-1)] then
   STOP; else choose one, $\alpha_{\min}$, randomly.
3. From among all implicants that cover $\alpha_{\min}$,
   choose the maximal implicant, I.
4. Include I in the solution.
5. f ← f - I
6. Go to step 2.
```

Choosing the maximal implicant

An implicant can be written as, $I = p\,{}^{i_1}x_1^{j_1}\,{}^{i_2}x_2^{j_2} \cdots {}^{i_n}x_n^{j_n}$.

Let the size of this implicant be the number of locations where it is p, which is in the range [1..(r-1)]. The size is,

$$\prod_{1 \le m \le n} (j_m - i_m + 1)$$

An implicant J is 'larger' than implicant I if the size of J is larger than the size of I. The maximal implicant is chosen as follows:

```
cur_size              ← - ∞
cur_terms_covered     ← - ∞
for each implicant I of f that completely covers
  $\alpha_{\min}$, do begin
    I_terms_covered ← the number of minterms in
    f that would be driven to 0 or r (don't care) if I
    is subtracted from f;

    I_size ← size of implicant I;

    if (I_terms_covered > cur_terms_covered) OR
      ((I_terms_covered = cur_terms_
      covered) AND (I_size > cur_size)) then begin
        best_implicant ← I;
        cur_size ← I_size;
        cur_terms_covered ← I_terms_covered;
    end;
end;
```

APPENDIX II

BESSLICH [3] ALGORITHM USING INTERVAL LITERALS

Algorithm:

Each location in orig_f corresponds to an assignment of values in the range [0..(r-1)] to the variables $x_1, x_2, \ldots, x_n$. The concatenation of these values is taken as an index which represents this location.

```
1. f ← orig_f
2. If f has no terms in the range [1...(r-1)], then
   STOP; else find the most isolated minterm
   $\alpha_{\min}$ in f.
3. From among all implicants that cover $\alpha_{\min}$,
   choose the maximal implicant, I.
4. Include I in the solution.
5. f ← f - I
6. Go to step 2.
```

Finding the most isolated minterm

Code the function values by mapping all 0's to -1; all r's to 0; and all other values to 1. Let cfn be the coded form of the function f.

Take the weight transform [3] of cfn and pick the minterm with the smallest weight as follows:

```
cur_lowest_wt ← ∞
for each term $\alpha$ in cfn such that $cfn(\alpha) = 1$ do
  begin

    /* compute the weight of this term with all
  other terms and add */

    $wt(\alpha) \leftarrow \sum_{\beta \in cfn} [cfn(\beta) * w(\alpha,\beta)]$

    if ($wt(\alpha) <$ cur_lowest_wt) then $\alpha_{\min} \leftarrow \alpha$;
end;
```

On exit from the for loop, $\alpha_{\min}$ is chosen as the most isolated minterm. In the loop, $w(\alpha,\beta)$ is computed as follows.

Let $i_n \, i_{n-1} \cdots i_2 \, i_1$ and $j_n \, j_{n-1} \cdots j_2 j_1$ represent the terms α and β. Then, compute

$$D(\alpha,\beta) = \sum_{1 \leq m \leq n} |i_m - j_m|$$

and $w(\alpha,\beta) = 2^{[n(r-1) - D(\alpha,\beta)]}$

Choosing the maximal implicant - Same as with Pomper and Armstrong

APPENDIX III

DUECK AND MILLER [6] ALGORITHM USING INTERVAL LITERALS

Algorithm:

1. f ← orig_f
2. If f has no terms in the range [1...(r-1)], then STOP; else find the most isolated minterm $\alpha_{\min}$ in f.
3. From among all implicants that cover $\alpha_{\min}$, choose the maximal implicant, I.
4. Include I in the solution.
5. f ← f - I, if f $\geq$ I; otherwise, I ← r.
6. Go to step 2.

Finding the most isolated minterm

Let min_val be the minimum value of all terms in f that are in the range 1...(r-1) i.e, neither 0 nor don't care. For each term α in the function with value equal to min_val, compute the clustering factor (CF) as follows -

$$CF(\alpha) = DEA_{\alpha}(r\text{-}1) + EA_{\alpha},$$

where EA_{α} is the number of minterms with which α can be combined in an interval literal and DEA_{α} is the number of variables (directions) in which α can be combined with a nonzero number of minterms.

Dueck and Miller actually define the isolation factor (IF) to be the reciprocal of the clustering factor above and pick the minterm with the highest IF. The same result can be obtained by picking the minterm with the smallest CF, avoiding the division. Ties are broken arbitrarily. Let $\alpha_{\min}$ be the minterm thus chosen.

Choosing the maximal implicant

```
cur_rbc ← ∞
for every valid implicant I in f that completely cov-
          ers $\alpha_{\min}$ do begin
    rbc ← calculated_rbc_of_I_in_f;
    if (rbc < cur_rbc) then begin
        best_implicant ← I;
        cur_rbc ← rbc;
    end;
end;
```

Calculation of rbc

```
rbc ← 0;
for each $\alpha$ in I such that $|\alpha| \neq r$ do begin
    along each dimension i = 1 to n do begin
        (if $|\alpha| \leq |$ I$|$) OR (if the neighbor $\beta$ immedi-
        ately before $\alpha$ in this dimension (if such $\beta$
        exists) is not in I and is such that $|\beta| = |\alpha|$
        - I) OR (if the neighbor $\beta$ immediately after
        $\alpha$ in this dimension (if such $\beta$ exists) is not
        in I and is such that $|\beta| = |\alpha|$ - I) then
            rbc ← rbc -1;
        (if the neighbor $\beta$ immediately before $\alpha$ in
        this dimension (if such $\beta$ exists) is not in I
        and is such that $|\beta| = |\alpha|$) OR (if the
        neighbor $\beta$ immediately after $\alpha$ in this
        dimension (if such $\beta$ exists) is not in I and is
        such that $|\beta| = |\alpha|$) then
            rbc ← rbc + 1;
    end;
end;
```